22｜RUM猜想：想要读写快还是存储省？又是三选二

2020-09-28 王磊

分布式数据库30讲 进入课程 >

讲述：王磊
时长 26:41　大小 24.44M

你好，我是王磊。

从第 18 讲，我们开始介绍查询过程中全部重要节点的相关技术，从并行框架到查询执行引擎，再从关联运算符到行式和列式存储。今天这一讲我们面临最后的一个步骤，直接和磁盘打交道，实现最终的数据存储，这就是存储引擎。

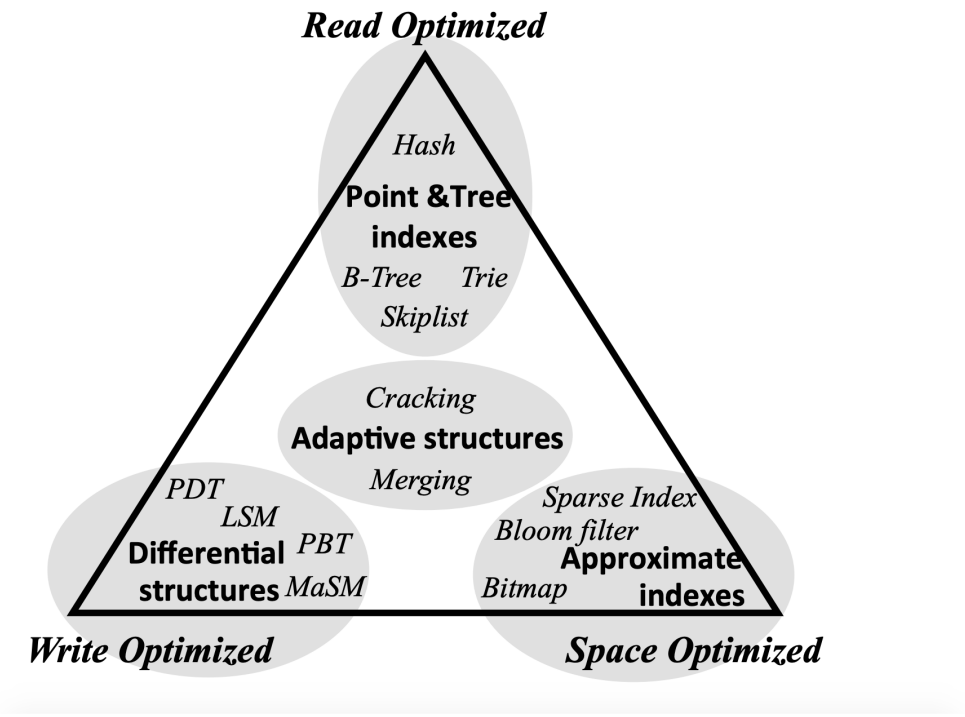
RUM 猜想

说到数据存储，我相信很多人都有一种直觉，那就是读写两种操作的优化在很多时候是斥的。

我们可以想一下，数据以什么形式存储，可以实现最快的写入速度？答案肯定是按照顺序写入，每次新增数据都追加在文件尾部，因为这样物理磁盘的磁头移动距离最小。

但这种方式对于读取显然是不友好的，因为没有任何有意义的数据结构做辅助，读操作必须从头到尾扫描文件。反过来，如果要实现更高效的读取，就要设计更复杂的数据结构，那么写入的速度当然就降低了，同时在存储空间上也会有额外的要求。

2016 年的一篇论文将我们这种朴素的理解提升到理论层面，这就是 RUM 猜想。RUM 猜想来自论文“Designing Access Methods: The RUM Conjecture”（Manos Athanassoulis et al.(2016)），同时被 SIGMOD 和 EDBT 收录。它说的是，对任何数据结构来说，在 Read Overhead（读）、Update Overhead（写） 和 Memory or Storage Overhead（存储） 中，同时优化两项时，需要以另一项劣化作为代价。论文用一幅图展示了常见数据结构在这三个优化方向中的位置，这里我摘录下来便于你理解。



引自Manos Athanassoulis et al.(2016)

在这张图中，我们可以看到两个非常熟悉的数据结构 B-Tree 和 LSM，它们被用于分布式数据库的存储引擎中，前者（实际是 B+Tree，B-Tree 的变体）主要用于 PGXC，后者则主要用于 NewSQL。这是不是代表 PGXC 就是要针对读操作，而 NewSQL 是针对写操作呢？并没有这么简单，还是要具体分析数据结构的使用过程。

下面，我们先从 B+Tree 说起，它也是单体数据库广泛使用的数据结构。

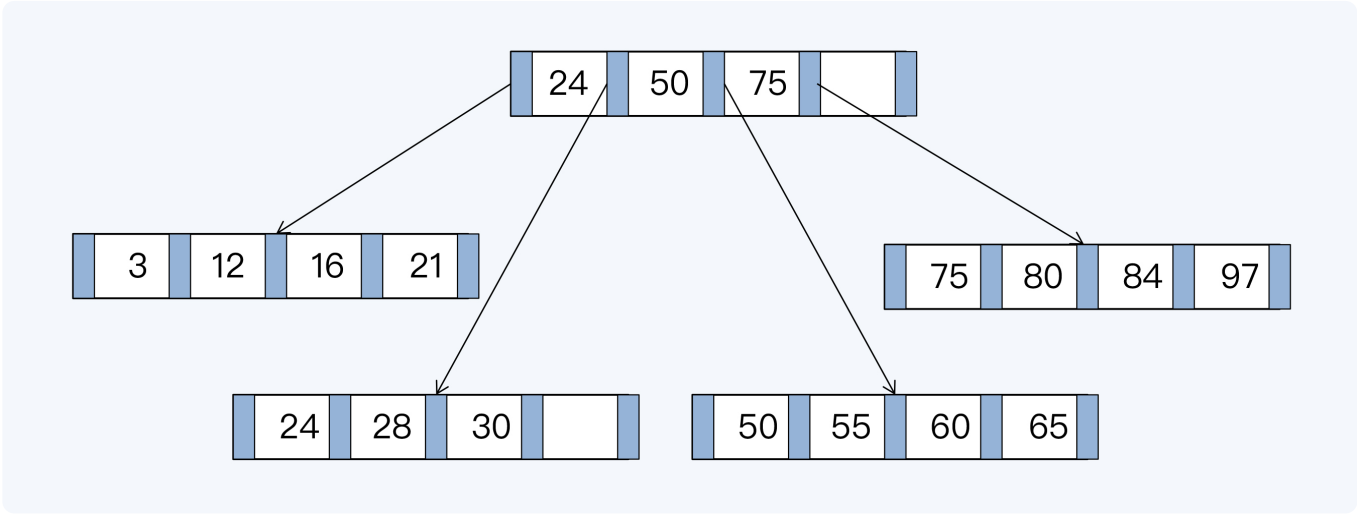
存储结构

B+Tree

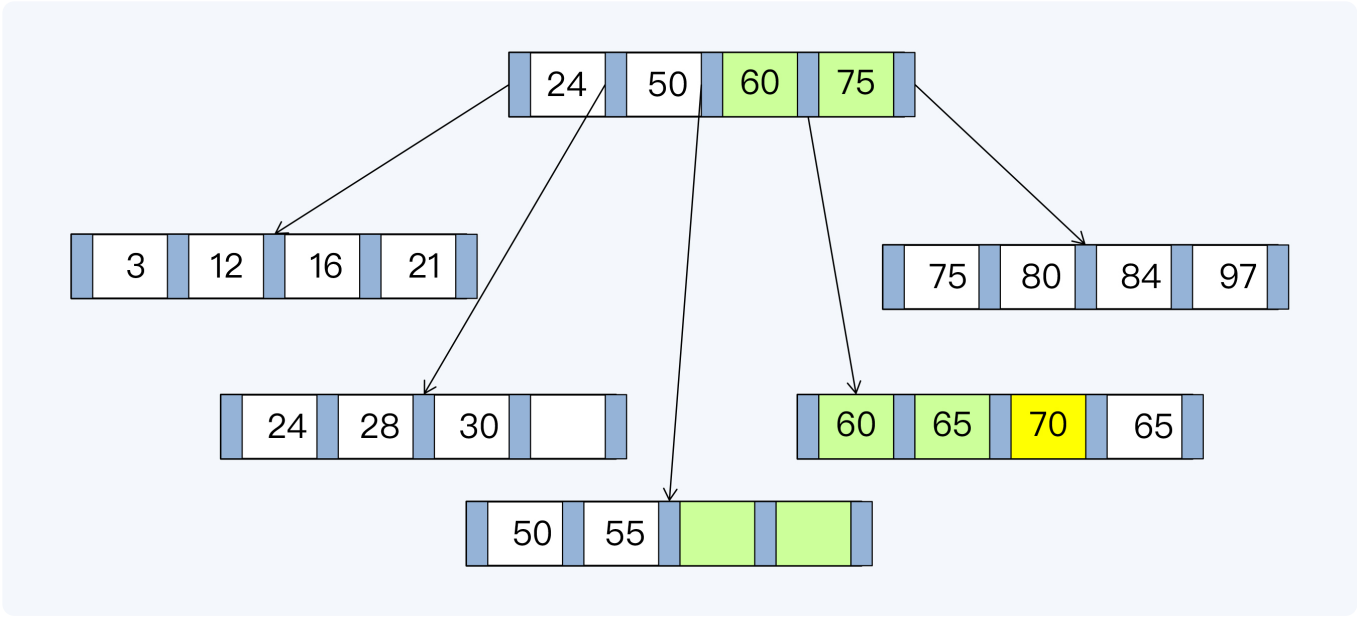
B+Tree 是对读操作优化的存储结构，能够支持高效的范围扫描，叶节点之间保留链接并且按主键有序排列，扫描时避免了耗时的遍历树操作。

我们用一个例子来演示下 MySQL 数据库的 B+Tree 写操作过程。

下面这张图中展示了一棵高度为 2 的 B+Tree，数据存储在 5 个页表中，每页可存放 4 条记录。为了方便你理解，我略去了叶子节点指向数据的指针以及叶子节点之间的顺序指针。



B+Tree 由内节点（InterNode）和叶节点（LeafNode）两类节点构成，前者仅包含索引信息，后者则携带了指向数据的指针。当插入一个索引值为 70 的记录，由于对应页表的记录已满，需要对 B+Tree 重新排列，变更其父节点所在页表的记录，并调整相邻页表的记录。完成重新分布后的效果如下：



在这个写入过程中存在两个问题：

1. 写放大

本来我们仅需要一条写入记录（黄色标注），实际上更新了 3 个页表中的 7 条索引记录，额外的 6 条记录（绿色标注）是为了维护 B+Tree 结构产生的写放大。

为了度量写放大的程度，相关研究中引入了写放大系数（Write Amplification Factor，WAF）这个指标，就是指实际写入磁盘的数据量和应用程序要求写入数据量之比。对于空间放大有类似的度量单位，也就是空间放大系数（Space Amplification Factor, SAF）。

我们这个例子中的 WAF 是 7。

2. 存储不连续

虽然新增叶节点会加入到原有叶节点构成的有序链表中，整体在逻辑上是连续的，但是在磁盘存储上，新增页表申请的存储空间与原有页表很可能是不相邻的。这样，在后续包含新增叶节点的查询中，将会出现多段连续读取，磁盘寻址的时间将会增加。

也就是说，虽然 B+Tree 结构是为读取做了优化，但如果进行大量随机写还是会造成存储的碎片化，从而导致写放大和读放大。

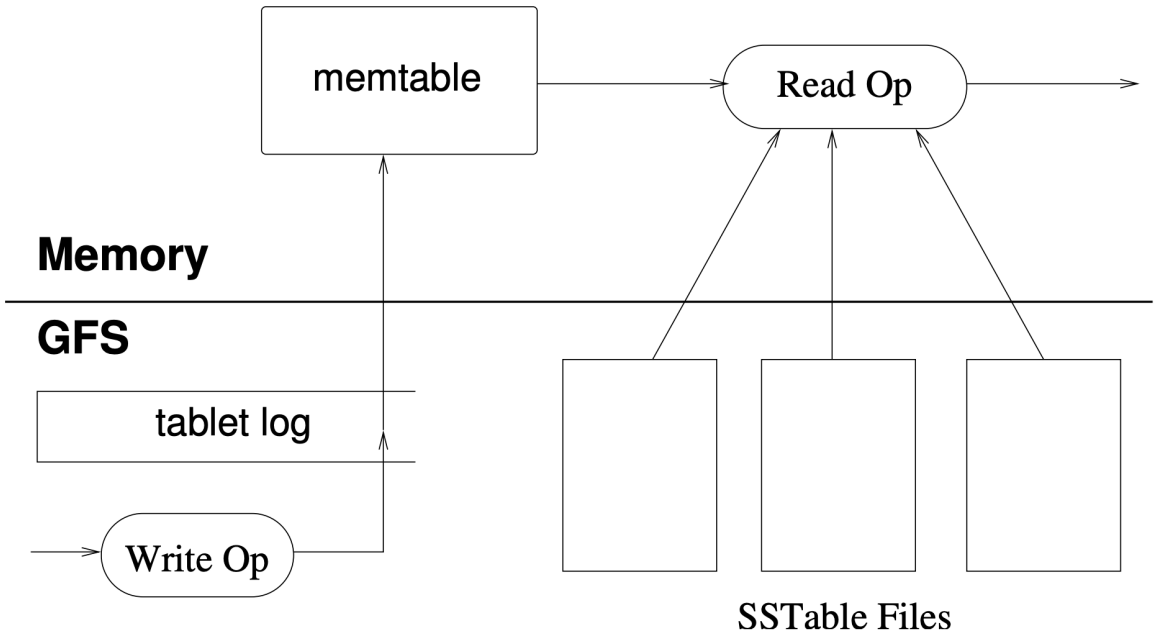
填充因子

填充因子（Factor Fill）是一种常见的优化方法，它的原理就是在页表中预留一些空间，这样不会因为少量的数据写入造成树结构的大幅变动。但填充因子的设置也很难拿捏，过大则无法解决写放大问题；过小会造成页表数量膨胀，增大对磁盘的扫描范围，降低查询性能。

相对于 PGXC，NewSQL 风格分布式数据库的底层存储引擎则主要采用 LSM-Tree。

LSM-Tree

LSM-Tree（Log Structured-Merge Tree）由 Patrick O’ Neil 在 1996 年的同名论文中首先提出。而后 Google 在 Bigtable（Fay Chang et al.(2008)）中使用了这个模型，它的大致处理过程如下图所示：

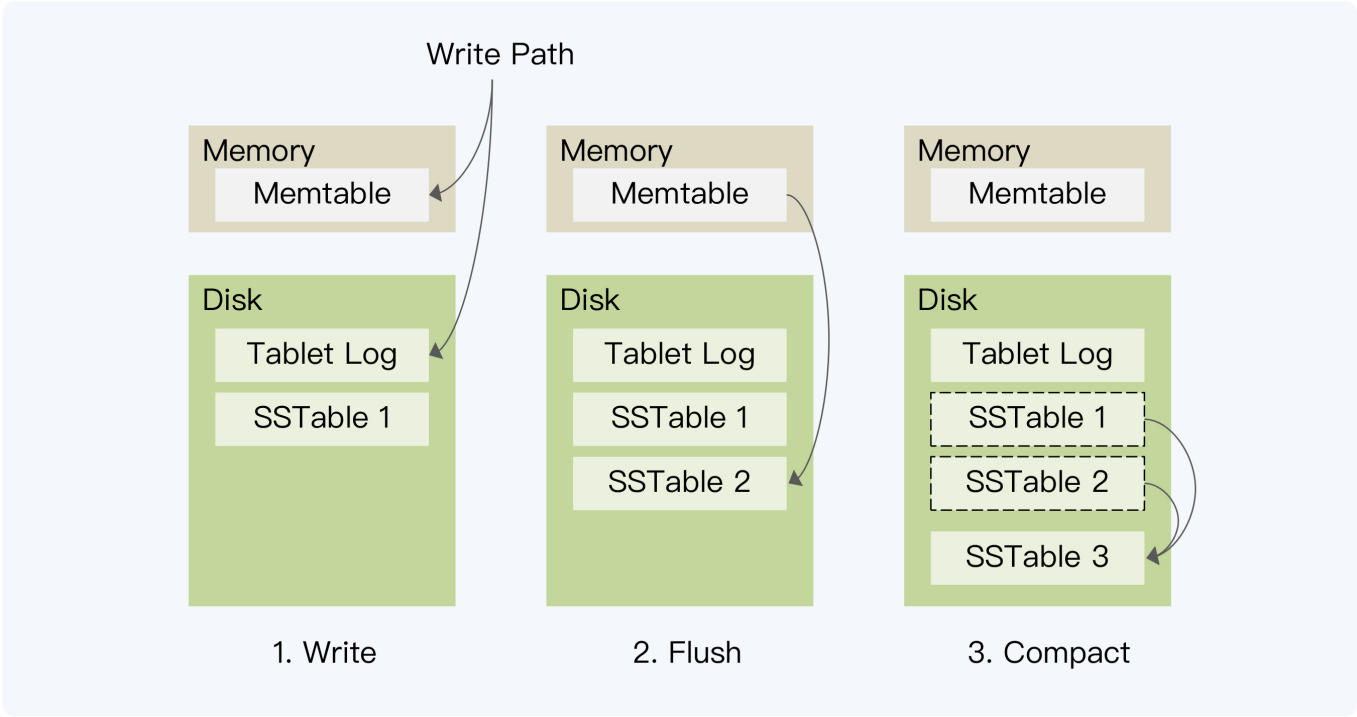


引自Fay Chang et al.(2008)

系统接收到写操作后会记录日志（Tablet Log）并将数据写入内存（Memtable），这时写操作就可以返回成功了。而在系统接收到读操作时，会在内存和磁盘文件中查找对应的数据。

你肯定会说不对呀，如果写入的数据都在内存中，磁盘文件又是哪来的呢？

的确，这中间还少了重要的一步，就是数据落盘的过程，这也是 LSM-Tree 最有特点的设计。其实，LSM 是分成三步完成了数据的落盘。



1. 第一步已经说过了，就是写入 Memtable，同时记录 Tablet Log；
2. 当 Memtable 的数据达到一定阈值后，系统会把其冻结并将其中的数据顺序写入磁盘上的有序文件（Sorted String Table，SSTable）中，这个操作被称为 Flush；当然，执行这个动作的同时，系统会同步创建一个新的 Memtable，处理写入请求。
3. 根据第二步的处理逻辑，Memtable 会周期性地产生 SSTable。当满足一定的规则时，这些 SSTable 会被合并为一个大的 SSTable。这个操作称为 Compact。

与 B+Tree 的最大不同是 LSM 将随机写转换为顺序写，这样提升了写入性能。另外，Flush 操作不会像 B+Tree 那样产生写放大。

我猜你会说，不对呀，Flush 是没有写放大，但还有 Compact 呢？

说的没错，真正的写放大就发生在 Compact 这个动作上。Compact 有两个关键点，一是选择什么时候执行，二是要将哪些 SSTable 合并成一个 SSTable。这两点加起来称为“合并策略”。

我们刚刚在例子中描述的就是一种合并策略，称为 Size-Tiered Compact Strategy，简称 Tiered。BigTable 和 HBase 都采用了 Tiered 策略。它的基本原理是，每当某个尺寸的 SSTable 数量达到既定个数时，将所有 SSTable 合并成一个大的 SSTable。这种策略的优点是比较直观，实现简单，但是缺点也很突出。下面我们从 RUM 的三个角度来分析下：

1. 读放大

执行读操作时，由于单个 SSTable 内部是按照 Key 顺序排列的，那么查找方法的时间复杂度就是 O(logN)。因为 SSTable 文件是按照时间间隔产生的，在 Key 的范围上就会存在交叉，所以每次读操作都要遍历所有 SSTable。如果有 M 个 SSTable，整体时间复杂度就是 O(M*logN)。执行 Compact 后，时间复杂度降低为 O(log(M*N))。在只有一个 SSTable 时，读操作没有放大。

2. 写放大

Compact 会降低读放大，但却带来更多的写放大和空间放大。其实 LSM 只是推迟了写放大，短时间内，可以承载大量并发写入，但如果是持续写入，则需要一部分 I/O 开销用于处理 Compact。

你可能听说到过关于 B+Tree 和 LSM 的一个争论，就是到底谁的写放大更严重。如果是采用 Tiered 策略，LSM 的写放大比 B+Tree 还严重。此外，Compact 是一个很重的操作，占用大量的磁盘 I/O，会影响同时进行的读操作。

3. 空间放大

从空间放大的角度看，Tiered 策略需要有两倍于数据大小的空间，分别存储合并前的多个 SSTable 和合并后的一个 SSTable，所以 SAF 是 2，而 B+Tree 的 SAF 是 1.33。

看到这你可能会觉得奇怪，LSM 好像也不是太优秀呀。只有短时间内写入速度快和阶段性的控制读放大这两个优势，在写放大和空间放大上都做得不好，而且还有 Compact 这种耗费 I/O 的重量级操作。那 LSM 为什么还这么流行呢?

这是因为 LSM 还有另一种合并策略，Leveled Compact Strategy，简称 Leveled 策略。

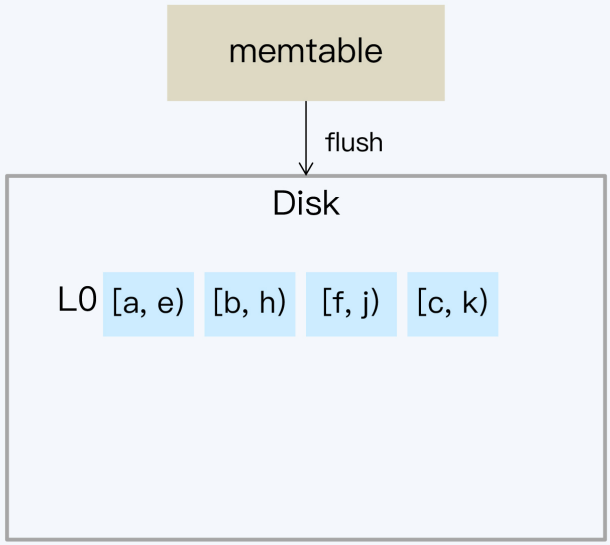
Leveled Compact Strategy

Tiered 策略之所以有严重的写放大和空间放大问题，主要是因为每次 Compact 需要全量数据参与，开销自然就很大。那么如果每次只处理小部分 SSTable 文件，就可以改善这个问题了。

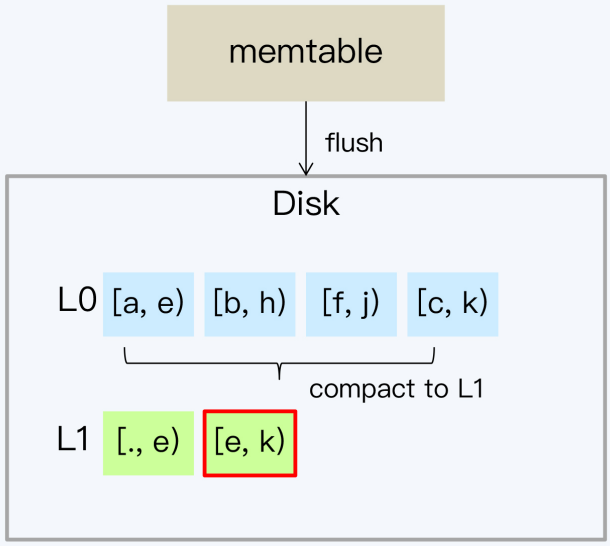
Leveled 就是这个思路，它的设计核心就是将数据分成一系列 Key 互不重叠且固定大小的 SSTable 文件，并分层（Level）管理。同时，系统记录每个 SSTable 文件存储的 Key 的范围。Leveled 策略最先在 LevelDB 中使用，也因此得名。后来从 LevelDB 上发展起来的 RocksDB 也采用这个策略。

接下来，我们详细说明一下这个处理过程。

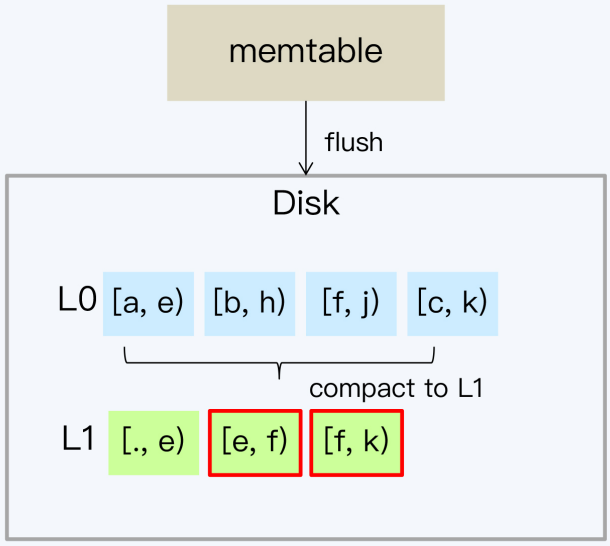
1. 第一步处理 Leveled 和 Tiered 是一样的。当内存的数据较多时，就会 Flush 到 SSTable 文件。对应内存的这一层 SSTable 定义为 L0，其他层的文件我们同样采用 Ln 的形式表示，n 为对应的层数。因为 L0 的文件仍然是按照时间顺序生成的，所以文件之间就可能有重叠的 Key。L0 不做整理的原因是为了保证写盘速度。



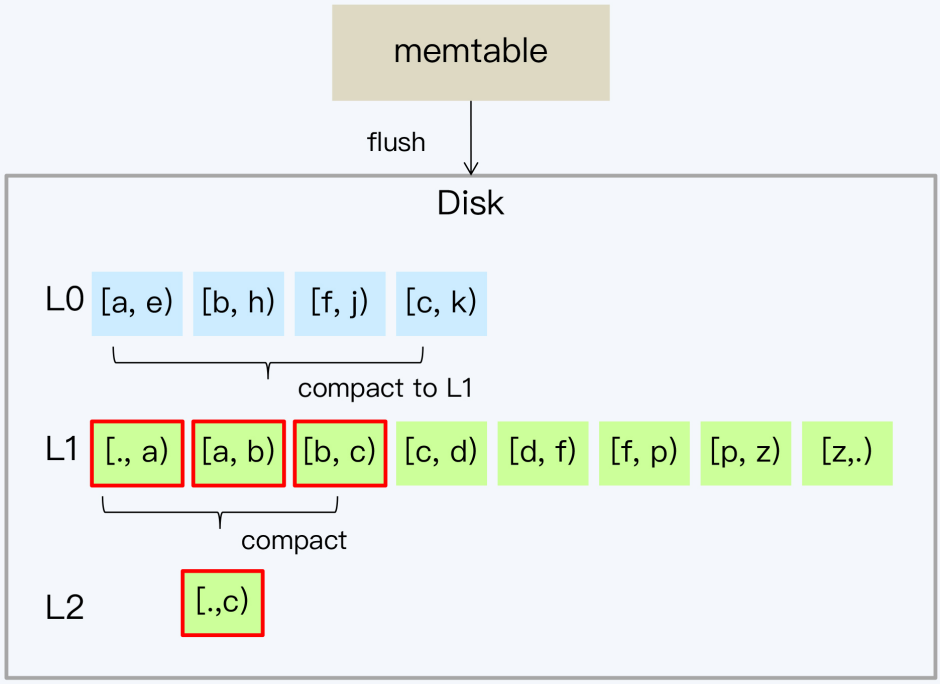
2. 通常系统会通过指定 SSTable 数量和大小的方式控制每一个层的数据总量。当 L0 超过预定文件数量，就会触发 L0 向 L1 的 Compact。因为在 L0 的 SSTable 中 Key 是交叉的，所以要读取 L0 的所有 SSTable，写入 L1，完成后再删除 L0 文件。从 L1 开始，SSTable 都是保证 Key 不重叠的。



3. 随着 L1 层数据量的增多，SSTable 可能会重新划分边界，目的是保证数据相对均衡的存储。



4. 由于 L1 的文件大小和数量也是受限的，所以随着数据量的增加又会触发 L1 向 L2 的 Compact。因为 L1 的文件是不重叠的，所以不用所有 L1 的文件都参与 Compact，这就延缓了磁盘 I/O 的开销。而 L2 的单个文件容量会更大，通常从 L1 开始每层的存储数据量以 10 倍的速度增长。这样，每次 Ln 到 L(n+1) 的 compact 只会涉及少数的 SSTable，间隔的时间也会越来越长。



说完处理流程，我们再从 RUM 三个角度来分析下。

1. 读放大

因为存在多个 SSTable，Leveled 策略显然是存在读放大的。因为 SSTable 是有序的，如果有 M 个文件，则整体计算的时间复杂度是 O(MlogN)。这个地方还可以优化。通常的方法是在 SSTable 中引入 Bloom Filter（BF），这是一个基于概率的数据结构，可以快速地确定一个 Key 是否存在。这样执行 Get 操作时，先读取每一个 SSTable 的 BF，只有这个 Key 存在才会真的去文件中查找。

那么对于多数 SSTable，时间复杂度是 O(1)。L0 层的 SSTable 无序，所有都需要遍历，而从 L1 层开始，每层仅需要查找一个 SSTable。那么优化后的整体时间复杂度就是 O(X+L-1+logN)，其中 X 是 L0 的 SSTable 数量，L 是层数。

2. 写放大

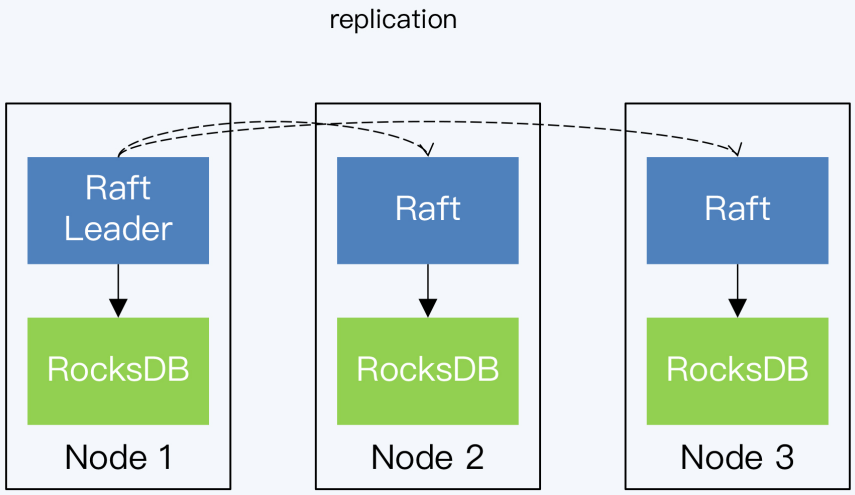
Leveled 策略对写放大是有明显改善的，除了 L0 以外，每次更新仅涉及少量 SSTable。但是 L0 的 Compact 比较频繁，所以仍然是读写操作的瓶颈。

3. 空间放大

数据在同层的 SSTable 不重叠，这就保证了同层不会存在重复记录。而由于每层存储的数据量是按照比例递增的，所以大部分数据会存储在底层。因此，大部分数据是没有重复记录的，所以数据的空间放大也得到了有效控制。

分布式数据库的实现

到这里我们已经介绍了存储结构的实现原理和它们在单体数据库中的运转过程。在分布式数据库中，数据存储是在每个数据节点上各自进行的，所以原理和过程是完全一样的。因此，TiDB 和 CockroachDB 干脆直接使用 RocksDB 作为单机存储引擎。在这两个分布式数据库的架构中，RocksDB 都位于 Raft 协议之下，承担具体的数据落地工作。



他们为什么选择 RocksDB 呢？CockroachDB 的官方解释是，他们的选择完全不是因为要考虑 LSM 的写优化能力，而是因为 RocksDB 作为一个成熟的单机存储引擎，可以加速 CockroachDB 的开发过程，而 RocksDB 本身也非常优秀，是他们能找到的最好选择。另一方面，TiDB 虽然没有提到选择一个写优化的存储引擎是否有特殊意义，但同样也认为 RocksDB 是一个非常优秀开源项目，可以满足他们对单机引擎的各种要求。

不过，很有意思的地方是，经过了早期版本的演进，TiDB 和 CockroachDB 不约而同都推出了自己的单机存储引擎。

你猜是什么原因呢？我在稍后再给出答案。现在让我们先看看另一款 NewSQL 风格分布式数据库 OceanBase 的实现。

OceanBase

OceanBase 选择了自研方式，也就没有直接引用 RocksDB，但它的存储模型基本上也是 LSM，也就同样需要面对 Compact 带来的一系列问题。我们来看看 OceanBase 是怎么优化的。

宏块与微块

在 Compact 过程中，被合并文件中的所有数据都要重写到新文件中。其实，对于那些没有任何修改的数据，这个过程是可以被优化的。

OceanBase 引入了宏块与微块的概念，微块是数据的组织单元，宏块则由微块组成。这样在进行 Compact 操作时，可以在宏块和微块两个级别判断，是否可以复用。如果在这两个级别数据没有发生实质变化，则直接进行块级别的拷贝，这样就省去了更细粒度的数据解析、编码以及计算校验和（Checksum）等操作。

轮转合并

OceanBase 还在多副本基础上设计了轮转合并机制。我们知道，根据 Raft 协议或者 Paxos 协议，总有多个副本同时存储着最新的数据，那么我们就可以用多副本来解耦 Compact 操作和同时段的查询操作，避免磁盘 I/O 上的竞争。

它的大致设计思路是这样的，将 Compact 操作放在与 Leader 保持数据同步的 Follower 上执行，而 Leader 节点则保持对外提供查询服务。当 Compact 完成后，改由那个 Follower 对外提供查询服务，Leader 和其他副本则去执行 Compact。

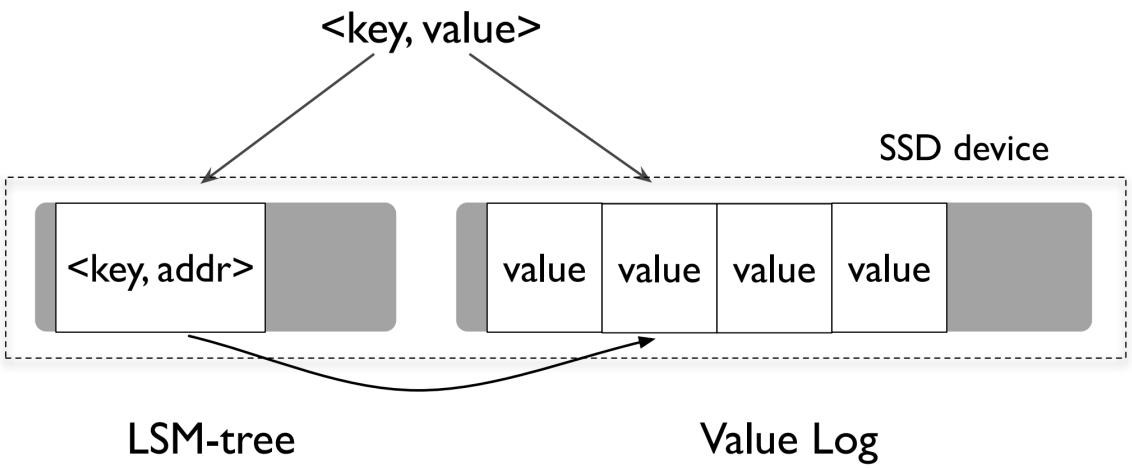
OceanBase 的这两项优化虽然没有降低写放大系数，但却有效减少了 Compact 过程中的 I/O 竞争。

TiDB：WiscKey

OceanBase 的优化还停留在工程层面，那么还有没有更好的理论模型呢?

2016 年真的出现了新的模型，这就是 WiscKey。论文“WiscKey: Separating Keys from Values in SSD-conscious Storage”（Lanyue Lu et al.(2016)）阐述了这种模型的设计思想。WiscKey 提出的改进是通过将 value 分离出 LSM-Tree 的方法来降低写放大。

WiscKey 的主要设计思想是，在 SSTable 的重写过程中，核心工作是对 Key 进行整理，保证多个 SSTable 的 Key 范围不重叠，且内部有序。而这个过程中 Value 的重写是没有太大价值的，而从实践看，Value 占用的存储空间远远大于 Key。这意味着大量的写操作和空间成本被浪费了。所以 WiscKey 提出将 Value 从 SSTable 中分离出来单独存储，这样就降低了写放大系数。



引自Lanyue Lu et al.(2016)

Value 单独存储的问题是按照 Key 连续读取数据时，对应的 Value 并不是连续存储，磁盘寻址成本增大。而 WiscKey 的设计前提是使用 SSD 替换 HDD，SSD 的随机读写效率接近于顺序读写，所以能够保持较高的整体效率。事实上，过高的写放大也会严重缩短 SSD 的使用寿命。WiscKey 就是针对 SSD 提出的存储模型。

说完 WiscKey，你或许猜到了 TiDB 和 Cockroach 放弃 RocksDB 的原因。

没错，TiDB 的新存储引擎 TiTan 就是受到 WiscKey 的启发，它目标之一就是将 Value 从 LSM-Tree 中分离出来单独存储，以降低写放大。

CockroachDB：Pebble

我们再来看看，CockroachDB 的单机存储引擎 Pebble。这里可能会让你有点意外，Pebble 并不是由于 WiscKey 的模型改进，它的出现完全是工程因素。首先是 Go 与 C 之间的调用障碍（CGo barrier）。

CGo barrier

CockroachDB 采用 Go 作为编程语言，而 RocksDB 是由 C++ 编写的。所以 CockroachDB 要面临 Go 到 C 的之间的调用障碍。测试表明，每一次对 RocksDB 的调用都要额外付出 70 纳秒的延迟。这个数字虽然看上去并不大，但是由于 K/V 操作非常频繁，总体成本仍然很可观。CockroachDB 也通过一些设计来降低延迟，比如将多次 K/V 操作合并成一次对 RocksDB 的操作，但是这一方面带来设计的复杂性，另外也无法从根本上解决问题。

值得一提的是，TiDB 也使用了 Go 语言作为主力开发语言，同样面临了这个问题。TiDB 最终是在底层存储 TiVK 放弃 Go 而选择 Rust 的，部分原因就是 Rust 与 C++ 之间的调用成本要低得多。

代码膨胀

CockroachDB 替换存储引擎的另一个原因是，RocksDB 的代码量日益膨胀。早期 RocksDB 的代码行数仅是 30k，而今天已经增加到 350k+。这很大程度是由于 RocksDB 的成功，很多软件选择了 RocksDB 作为存储引擎，包括 MySQL（MyRocks）、Flink 等，这驱动 RocksDB 增加了更丰富的特性，也导致体量越来越大。但这些丰富的特性对 CockroachDB 来说并不是必须的，反而引入了不必要的变更风险。而 Cockraoch 开发的 Pebble，代码行数则仅有 45k，的确是小巧得多了。

所以，总的来说，CockraochDB 替换存储引擎是工程原因，而其中 CGO barrier 的这个问题更像是在偿还技术债。

TiFlash

到这里，分布式数据库下典型的存储引擎就介绍完了，它们都适用于 OLTP 场景，要求在小数据量下具有高效的读写能力。而 OLAP 下的存储引擎则有很大的不同，通常不会对单笔写入有太高的要求，但也有例外，还记得我在第 18 讲提到过的 OLAP 存储引擎 TiFlash 吗？由于实时分析的要求，它必须及时落盘来自 TiKV 的数据，同样需要很高的写入速度。

在 18 讲中并没对 TiFlash 展开说明，我想，现在你应该能够猜出 TiFlash 的关键设计了吧。是的，高效写入的秘密就在于它的存储模型 Delta Tree 采用了类似 LSM 的结构。其中的 Delta Layer 和 Stable Layer，分别对应 LSM Tree 的 L0 和 L1，Delta Layer 可以顺序写入。

小结

那么，今天的课程就到这里了，让我们梳理一下这一讲的要点。

1. RUM 猜想提出读负载、写负载、空间负载这三者之间只能优化两个目标，另外一个目标只能被弱化。在典型数据结构中，B+Tree 是读取优化，LSM 是写入优化，这两种存储结构正好对应了两种风格分布式数据库的存储引擎。
2. PGXC 和单体数据库采用了 B+Tree，随机写会带来页表分裂和存储不连续，导致写放大和读放大。常用的优化方式是通过填充因子预留页空间。LSM 将随机写转换为顺序写提升了写入速度，但只是延缓了写放大并没有真正减少写放大。如果采用 Tiered 策略，LSM 的写放大和空间放大都高于 B+Tree。不过，LSM 可以很好地控制读放大，读操作的时间复杂度是 O(logN)。
3. Level 策略，降低了写放大和空间放大，同时读操作的成本也没有大幅增加，是一个比较广泛使用的存储模型。RocksDB 采用了这种存储模型，加上优秀的工程实现，被很多软件引用。TiDB 和 CockroachDB 也使用 RocksDB 作为单机存储引擎。OceanBase 在工程层面对 LSM 进行了优化。
4. WiscKey 是对 LSM 的理论模型优化，通过 Key/Value 分离存储的方式降低写放大，但这种设计会增加随机读写，要以使用 SSD 为前提。TiDB 受到 WiscKey 的启发，自研了 TiTan。
5. CockroachDB 也推出了自己的存储引擎 Pebble，主要目的是解决工程问题，包括解决 CGo barrier 问题和避免 Rust 功能膨胀带来的变更风险。

这一讲，我们简单介绍了近年来的主流存储模型，不难发现 LSM 占据了主导位置。但这并不代表 B+Tree 是一种失败的设计，两者只是在 RUM 之间选择了不同的优化方向。同时，是否有可靠的开源工程实现也直接影响到了对应存储模型的流行度，这一点和 Raft 协议的大规模使用很相似。

我的一点建议是，记下这些存储模型虽然重要，但更加关键的是对 RUM 猜想的深入思考，因为架构设计的精髓就在于做出适当的取舍。

思考题

课程的最后，我们来看看今天的思考题。今天的课程中，我们介绍了不同的存储模型，其中也提到了 Bloom Filter 的使用。它是一种神奇的数据结构，在 RUM 中是空间优化的典型，用很少的空间就可以记录很多数值是否存在。这种数据结构提供了两种操作方式，用于数值的写入和数值的检核，但数据库查询更多是范围查询，在 K/V 中体现为 Scan 操作，显然不能直接对应。所以我的问题是，Scan 操作是否可以使用 Bloom Filter 来加速，如果可以又该如何设计呢?

欢迎你在评论区留言和我一起讨论，我会在答疑篇和你继续讨论这个问题。如果你身边的朋友也对存储引擎这个话题感兴趣，你也可以把今天这一讲分享给他，我们一起讨论。

学习资料

Fay Chang et al.: *Bigtable: A Distributed Storage System for Structured Data*

Lanyue Lu et al.: *WiscKey: Separating Keys from Values in SSD-conscious Storage*

Manos Athanassoulis et al: *Designing Access Methods: The RUM Conjecture*

Patrick O’ Neil et al.: *The Log-Structured Merge-Tree (LSM-Tree)*

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 21 | 查询执行引擎：如何让聚合计算加速?

下一篇 23 | 数据库查询串讲：重难点回顾+思考题答疑+知识全景图

精选留言 (7)

写留言

piboye
2020-09-28

即使不是前缀相同，范围查询也可以一个一个key去尝试，bf是内存操作，还是比磁盘快吧

作者回复: 首先范围查询可能是无法穷举的，例如对浮点数的范围查询。另外即使可以穷举，每个都查的时间复杂度就是O(N)了，而磁盘文件是有序的，加上磁盘会做预取，可能还是磁盘快些。

1

Jxin
2020-09-28

课后题：
这样问，那就一定是可以了。但很遗憾，能力有限想不到。scan了的话，如果是常规的行式存储,bf起不到作用，毕竟边界不存在不等于边界内部不存在。

22讲下来的感觉:...
展开

作者回复: 嗯，谢谢，你的建议很中肯。因为这个专栏是中立视角，希望跳出某个具体的产品，讲一些共性的、原理层面的研究进展，然后选择有代表性的产品做一些映射。这样一来，所有的内容就不能与某一款产品完全匹配，会有些抽象，但我相信后面一定会有更多分布式数据库的课程作为补充，做到虚实结合。

1

扩散性百万咸面包
2020-09-28

思考题：
使用Bloom Filter的改造版，现在不用来判断某个key是否不存在，而判断某个key的前缀

是否不存在。比如说我要查300-000 到 300-999的区间，那么他们拥有共同前缀300，我只要判断这个前缀是不是在当前SSTable里即可。

展开

作者回复: 说的全中，点赞。

1

licl1008
2020-09-28

想问一下 写放大和空间放大具体有啥区别 写得多了不是磁盘空间就占用更多了吗?

piboye
2020-09-28

lsm最核心的是顺序写特性，即使写放大和b+一样大，因为顺序写比随机写还是快很多。写放大和空间放大比较影响ssd的寿命。

展开

有铭
2020-09-28

为什么RUM是个猜想而不是个定论呢？这东西第一眼看上去好像CAP理论，不过CAP已经被证明了

可怜大灰狼
2020-09-28

数据块里是按照顺序存储了很多条数据。我想对数据块建立一级布隆过滤器实例，这样scan定位起始和终止对应的数据块，应该能提速不少。
如果把数据块按照大小分级，类似LSM的level层那样。是不是会更快些?

展开

作者回复: 说的很好，点赞。